Kernel testing frameworks

Brendan Higgins
Shuah Khan

Why kselftest?

- Regression test suite
- Focuses on testing kernel from user-space
- User-space applications (Shell scripts, C programs)
  - Kernel Test modules used to exercise kernel code paths
- Allows for breadth and depth coverage (error paths etc.)
- Not for workload or application testing

Why kselftest?

- Perfect for feature, functional and regression testing
- Perfect for bug fix focused regression testing and subsystem testing
- Perfect for testing user APIs, system calls, critical user paths, common use cases
- Perfect for end to end regression testing
  - Provides assurance that “everything works”
- Combination of Open and Closed box testing
- For more information on Kselftest framework/run/write tests
  - Watch LF Live Mentorship webinar:
    - Kernel Validation With Kselftest

Why KUnit?

- Focuses on in-kernel testing
- Perfect for:
  - testing internal kernel APIs
  - libraries, drivers, …,
  - individual *units* of code
- Perfect for unit testing
  - Makes it tractable to test all the edge cases

McCabe's Complexity

- Testing all edge cases?
  - Imagine trying to reach an arbitrary edge case in the kernel from a syscall
  - Reaching every state is intractable
- Solution: Call functions directly to test edge cases

McCabe's Complexity

- Solution: Call functions directly to test edge cases
- McCabe's complexity is a measure of the number of states, or branches a function can achieve
- If we have a function, A, call other functions, B1, B2, …, Bn, and we only test A
  - If we try to reach all branches from A, you can see that as the function depth increases, the total number of branches increases combinatorially
  - If we only reach all the states of each function individually, the branches increase linearly.
- KUnit is a really practical way to test the vast majority of edge cases.

McCabe's Complexity

- Solution: Call functions directly to test edge cases
- McCabe's complexity is a measure of the number of states, or branches a function can achieve
- If we have a function, A, call other functions, B1, B2, …, Bn, and we only test A
  - If we try to reach all branches from A, you can see that as the function depth increases, the total number of branches increases combinatorially
  - If we only reach all the states of each function individually, the branches increase linearly.
- KUnit is a really practical way to test the vast majority of edge cases.
- For more background info on KUnit like this please see LF Live Mentorship webinar: KUnit Testing Strategies

GCOV: How to coverage

- GCOV keeps track of code run during execution
- Generates reports
- Show what code ran, and what code did not

```
        1 : static int apply_constraint(struct dev_pm_qos_request *req,
          :                             enum pm_qos_req_action action, s32 value)
          : {
        1 :         struct dev_pm_qos *qos = req->dev->power.qos;
          :         int ret;
          :
        1 :         switch(req->type) {
        1 :         case DEV_PM_QOS_RESUME_LATENCY:
        1 :                 if (WARN_ON(action != PM_QOS_REMOVE_REQ && value < 0))
        0 :                         value = 0;
          :
        1 :                 ret = pm_qos_update_target(&qos->resume_latency,
          :                                            &req->data.pnode, action, value);
        1 :                 break;
        0 :         case DEV_PM_QOS_LATENCY_TOLERANCE:
        0 :                 ret = pm_qos_update_target(&qos->latency_tolerance,
          :                                            &req->data.pnode, action, value);
        0 :                 if (ret) {
        0 :                         value = pm_qos_read_value(&qos->latency_tolerance);
        0 :                         req->dev->power.set_latency_tolerance(req->dev, value);
          :                 }
          :                 break;
        0 :         case DEV_PM_QOS_MIN_FREQUENCY:
          :         case DEV_PM_QOS_MAX_FREQUENCY:
        0 :                 ret = freq_qos_apply(&req->data.freq, action, value);
        0 :                 break;
        0 :         case DEV_PM_QOS_FLAGS:
        0 :                 ret = pm_qos_update_flags(&qos->flags, &req->data.flr,
          :                                           action, value);
        0 :                 break;
          :         default:
          :                 ret = -EINVAL;
          :         }
          :
        1 :         return ret;
          : }
```

GCOV: How to coverage

- Shows directory level summaries

			Hit	Total	Coverage
Current view:	top level - drivers/base/power				
Test:	coverage.info	Lines:	234	2752	8.5 %
Date:	2021-09-20 14:11:03	Functions:	24	274	8.8 %

Filename	Line Coverage		Functions	
common.c	0.0 %	0 / 43	0.0 %	0 / 8
generic_ops.c	0.0 %	0 / 73	0.0 %	0 / 22
main.c	3.8 %	31 / 810	6.8 %	4 / 59
power.h	81.8 %	9 / 11	100.0 %	1 / 1
qos-test.c	100.0 %	49 / 49	100.0 %	3 / 3
qos.c	18.2 %	69 / 379	17.2 %	5 / 29
runtime.c	8.3 %	55 / 665	12.2 %	6 / 49
sysfs.c	5.2 %	13 / 248	5.7 %	2 / 35
wakeirq.c	0.0 %	0 / 99	0.0 %	0 / 11
wakeup.c	1.0 %	3 / 302	2.4 %	1 / 41
wakeup_stats.c	6.8 %	5 / 73	12.5 %	2 / 16

GCOV: How to coverage

- Shows directory level summaries

Current view:	top level - drivers/base/power			
		Hit	Total	Coverage
Test:	coverage.info			
	Lines:	234	2752	8.5 %
Date:	2021-09-20 14:11:03			
	Functions:	24	274	8.8 %

Filename	Line Coverage		Functions	
common.c	0.0 %	0 / 43	0.0 %	0 / 8
generic_ops.c	0.0 %	0 / 73	0.0 %	0 / 22
main.c	3.8 %	31 / 810	6.8 %	4 / 59
power.h	81.8 %	9 / 11	100.0 %	1 / 1
qos-test.c	100.0 %	49 / 49	100.0 %	3 / 3
qos.c	18.2 %	69 / 379	17.2 %	5 / 29
runtime.c	8.3 %	55 / 665	12.2 %	6 / 49
sysfs.c	5.2 %	13 / 248	5.7 %	2 / 35
wakeirq.c	0.0 %	0 / 99	0.0 %	0 / 11
wakeup.c	1.0 %	3 / 302	2.4 %	1 / 41
wakeup_stats.c	6.8 %	5 / 73	12.5 %	2 / 16

GCOV: How to coverage

- Shows directory level summaries
- Shows overall summary as coverage number

Current view: top level - drivers/base/power
Test: coverage.info
Date: 2021-09-20 14:11:03

	Hit	Total	Coverage
Lines:	234	2752	8.5 %
Functions:	24	274	8.8 %

Filename	Line Coverage			Functions	
common.c		0.0 %	0 / 43	0.0 %	0 / 8
generic_ops.c		0.0 %	0 / 73	0.0 %	0 / 22
main.c		3.8 %	31 / 810	6.8 %	4 / 59
power.h		81.8 %	9 / 11	100.0 %	1 / 1
qos-test.c		100.0 %	49 / 49	100.0 %	3 / 3
qos.c		18.2 %	69 / 379	17.2 %	5 / 29
runtime.c		8.3 %	55 / 665	12.2 %	6 / 49
sysfs.c		5.2 %	13 / 248	5.7 %	2 / 35
wakeirq.c		0.0 %	0 / 99	0.0 %	0 / 11
wakeup.c		1.0 %	3 / 302	2.4 %	1 / 41
wakeup_stats.c		6.8 %	5 / 73	12.5 %	2 / 16

Code Coverage **IS**

- A great way to quickly find what code **IS** tested and what code **IS NOT** tested.
- Allows you to quickly identify problem areas, and drill down into a report.
- Identify missed branches.
- Identify unused code.

Code Coverage **IS**: Example

- Imagine we are testing some code:

```
static int __dev_pm_qos_add_request(struct device *dev,
                                    struct dev_pm_qos_request *req,
                                    enum dev_pm_qos_req_type type, s32 value)
{
        int ret = 0;

        if (!dev || !req || dev_pm_qos_invalid_req_type(dev, type))
                return -EINVAL;

        if (WARN(dev_pm_qos_request_active(req),
                 "%s() called for already added request\n", __func__))
                return -EINVAL;

        if (IS_ERR(dev->power.qos))
                ret = -ENODEV;
        else if (!dev->power.qos)
                ret = dev_pm_qos_constraints_allocate(dev);

        trace_dev_pm_qos_add_request(dev_name(dev), type, value);
        if (ret)
                return ret;

        req->dev = dev;
        req->type = type;
        if (req->type == DEV_PM_QOS_MIN_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MIN, value);
        else if (req->type == DEV_PM_QOS_MAX_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MAX, value);
        else
                ret = apply_constraint(req, PM_QOS_ADD_REQ, value);

        return ret;
}
```

Code Coverage **IS**: Example

- Imagine we are testing some code:
- We can see that we have edge cases for
  - DEV_PM_QOS_MIN_FREQUENCY
  - DEV_PM_QOS_MAX_FREQUENCY
-

```
static int __dev_pm_qos_add_request(struct device *dev,
                                    struct dev_pm_qos_request *req,
                                    enum dev_pm_qos_req_type type, s32 value)
{
        int ret = 0;

        if (!dev || !req || dev_pm_qos_invalid_req_type(dev, type))
                return -EINVAL;

        if (WARN(dev_pm_qos_request_active(req),
                 "%s() called for already added request\n", __func__))
                return -EINVAL;

        if (IS_ERR(dev->power.qos))
                ret = -ENODEV;
        else if (!dev->power.qos)
                ret = dev_pm_qos_constraints_allocate(dev);

        trace_dev_pm_qos_add_request(dev_name(dev), type, value);
        if (ret)
                return ret;

        req->dev = dev;
        req->type = type;
        if (req->type == DEV_PM_QOS_MIN_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MIN, value);
        else if (req->type == DEV_PM_QOS_MAX_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MAX, value);
        else
                ret = apply_constraint(req, PM_QOS_ADD_REQ, value);

        return ret;
}
```

Code Coverage **IS**: Example

- Imagine we are testing some code:
- We can see that we have edge cases for
  - DEV_PM_QOS_MIN_FREQUENCY
  - DEV_PM_QOS_MAX_FREQUENCY
- The report shows us that our tests do not cover these edge cases.

```
static int __dev_pm_qos_add_request(struct device *dev,
                                    struct dev_pm_qos_request *req,
                                    enum dev_pm_qos_req_type type, s32 value)
{
        int ret = 0;

        if (!dev || !req || dev_pm_qos_invalid_req_type(dev, type))
                return -EINVAL;

        if (WARN(dev_pm_qos_request_active(req),
                 "%s() called for already added request\n", __func__))
                return -EINVAL;

        if (IS_ERR(dev->power.qos))
                ret = -ENODEV;
        else if (!dev->power.qos)
                ret = dev_pm_qos_constraints_allocate(dev);

        trace_dev_pm_qos_add_request(dev_name(dev), type, value);
        if (ret)
                return ret;

        req->dev = dev;
        req->type = type;
        if (req->type == DEV_PM_QOS_MIN_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MIN, value);
        else if (req->type == DEV_PM_QOS_MAX_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MAX, value);
        else
                ret = apply_constraint(req, PM_QOS_ADD_REQ, value);

        return ret;
}
```

Code Coverage **IS**: Example

- Imagine we are testing some code:
- We can see that we have edge cases for
  - `DEV_PM_QOS_MIN_FREQUENCY`
  - `DEV_PM_QOS_MAX_FREQUENCY`
- The report shows us that our tests do not cover these edge cases.
- This shows the power of KUnit with coverage
  - We can (and do) call this function directly in tests

```
static int __dev_pm_qos_add_request(struct device *dev,
                                    struct dev_pm_qos_request *req,
                                    enum dev_pm_qos_req_type type, s32 value)
{
        int ret = 0;

        if (!dev || !req || dev_pm_qos_invalid_req_type(dev, type))
                return -EINVAL;

        if (WARN(dev_pm_qos_request_active(req),
                 "%s() called for already added request\n", __func__))
                return -EINVAL;

        if (IS_ERR(dev->power.qos))
                ret = -ENODEV;
        else if (!dev->power.qos)
                ret = dev_pm_qos_constraints_allocate(dev);

        trace_dev_pm_qos_add_request(dev_name(dev), type, value);
        if (ret)
                return ret;

        req->dev = dev;
        req->type = type;
        if (req->type == DEV_PM_QOS_MIN_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MIN, value);
        else if (req->type == DEV_PM_QOS_MAX_FREQUENCY)
                ret = freq_qos_add_request(&dev->power.qos->freq,
                                           &req->data.freq,
                                           FREQ_QOS_MAX, value);
        else
                ret = apply_constraint(req, PM_QOS_ADD_REQ, value);

        return ret;
}
```

Code Coverage **IS NOT**

- Code coverage is a tool, not a panacea
- Code coverage helps quickly identify and prioritize problem areas
- Code coverage summaries **do not tell you whether your testing is good or bad**
  - What is the right amount of line coverage?
  - 50%?
  - 70%?
  - 90%?
  - 100%?

What's the right coverage?

- How do we measure coverage?
  - % of lines?
  - % of functions?
  - % of branches?
- What about absolute vs incremental?

What's the right coverage?

- Absolute coverage:
  - What you expect.
  - Everything in the entire codebase at some point in time.
- Incremental coverage:
  - The test coverage of the Δ in a change

Absolute vs. Incremental Coverage

- Incremental Coverage is usually more interesting
  - It’s much easier to achieve high incremental coverage immediately
  - Helps prioritize code more likely to be buggy
  - More actionable by developers
  - Code that has not changed in a long time is *more* likely to be fine

Absolute vs. Incremental Coverage

- Absolute Coverage is still important, just less important
  - Old code may be less likely to contain bugs…
  - ...but it's often worse when it does
- Often easier for comparing coverage health of subsystems
- Easier to compute

Kselftest & KUnit

- Kselftest
  - Good for depth testing covering deeper code paths
  - Good for testing commonly used code paths
  - A good test could test some error paths
- KUnit
  - Good for targeting error paths & edge cases
  - Easier and faster for zeroing in on a kernel area

Kernel Dependability - Safety critical space

- Code coverage important for Safety?
- Kselftest & KUnit
  - Improvements that could be made?
  - More tests for coverage?
  - More tests for regression?
  - ????